TeachScheme, ReachJava: Introducing OOP without Drowning in Syntax

Stephen Bloch, Adelphi University, Garden City, NY
John Clements, Cal Poly, San Luis Obispo, CA
Kathi Fisler, WPI, Worcester, MA
Matthew Flatt, University of Utah, Salt Lake City, UT
Viera K. Proulx, Northeastern University, Boston, MA
Supported by NSF grant #0618543
What is CS1 about?
What you teach

blah algorithm blah variable blah function
blah data type blah object blah re-use blah
methodology blah testing blah design
blah composition blah refinement blah
abstraction blah
What they hear

blah ; blah { blah ) blah ] blah return blah
public static void main (String[] args)
blah /* blah */ blah <int> blah if ( blah
JOptionPane blah class Posn(int x,y)
Programming languages: a necessary evil

• Necessary: to write real programs that really run on real computers
• Evil: distract students from the important stuff
• Will be obsolete in a few years anyway
How to minimize language?

- Introduce language features *one at a time*
- Avoid "black magic"; never ask students to write anything they can't understand *now*
- Introduce language features *only as necessary to teach an important concept*
- Corollary: any language feature that doesn't help teach a CS1 concept shouldn't be mentioned in CS1
- Corollary: need *enforced language subsets* so students don't accidentally use language features they haven't seen
- For GUI, need interface between student code (in subset) and library code (in full language)
Choosing a first language

![Graph showing time in course vs syntax rules for Java and C++]
Choosing a first language

![Graph showing syntax rules over time in course for Java, C++, and Scheme.](chart.png)
But what about OOP?

- Students need to learn OOP, and an OO language, by 2nd or 3rd semester
- OOP is a terrific way to organize multi-KLOC programs
- First-term students don't write multi-KLOC programs
- OOP's benefits aren't apparent in first term; students see only the overhead
- (Challenge: write a short, OO C++/Java program that wouldn't be simpler without OO)
How to respond to this?

• “I learned OOP as an add-on to procedural/imperative programming, so that’s the natural way to do it.”

… or …

• “Let’s teach the ideas that will be important in OOP, with as little syntactic & semantic overhead as possible.”
Gentle introduction to OOP

Same concepts that come up in OOP, but without syntactic burden of Java, C++, etc.

More time to teach good habits of analysis, design, testing, etc.

Switch to Java after 2-4 months: concepts familiar, only need to learn new syntax for them
An example

• A Shape is either a Circle or a Rectangle.
• A Circle has a center (of type Posn) and a radius (of type number).
• A Rectangle has a top-left (of type Posn), a width and a height (both of type number).
• Define a function which tells whether a given Shape contains a given Posn.
What's important in this assignment?

- Defining functions
- Data types with fields
- Variant data types (aka polymorphism)
- Testing; choice of test cases
The Shapes problem in Java

```java
interface Shape {
    boolean contains(Posn p);
}

class Rectangle implements Shape {
    private Posn topLeft;
    private double width, height;
    public Rectangle (Posn tl, double w, double h) {
        this.topLeft = tl;
        this.width = w;
        this.height = h;
    }
    public boolean contains(Posn p) {
        return p.getX() > this.topLeft.getX() &&
               p.getY() > this.topLeft.getY() &&
               p.getX() < this.topLeft.getX() + this.width &&
               p.getY() < this.topLeft.getY() + this.height;
    }
}

class Circle implements Shape {
    private Posn center;
    private double radius;
    public Circle(Posn c, double r) {
        this.center = c;
        this.radius = r;
    }
    public boolean contains(Posn p) {
        double dx = this.center.getX() - p.getX();
        double dy = this.center.getY() - p.getY();
        double dist = Math.sqrt(dx*dx + dy*dy);
        return dist < this.radius;
    }
}

Class TestShapes extends junit.framework.TestCase {
    public void testCircle () {
        Circle c = new Circle (new Posn(10,5), 5);
        assertTrue (c.contains (new Posn(10,5));
        assertTrue (c.contains (new Posn(11,3));
        assertFalse (c.contains (new Posn(6,9));
        assertFalse (c.contains (new Posn(6,8));
    }
    public void testRect () {
        Rectangle r = new Rectangle (new Posn(3,5), 7, 3);
        // 17 test cases omitted
    }
}
```
What was important?

+-------+
| Shape |
+-------+

+-------+
|       |
/ \    
|     |
+-----+

+---------------+
| Circle        |
+---------------+
| Posn center   |
| double r      |
+---------------+

+---------------+
| Rectangle     |
+---------------+
| Posn topLeft  |
| double width  |
| double height |
+---------------+
What was important?

• Defining functions
• Data types with fields
• Variant data types
• Testing; choice of test cases

A lot of the code isn’t about these things!
What's wrong with this program?

• Syntax for stuff that isn't used in this program, but *might* be used in *some* program *somewhere*.

• Much of it still required even for "Hello world".

• Students have to use language features they don't yet need or understand. Black magic! "Do this; you'll understand it in a month."
In our approach…

- The solution uses only concepts that are relevant to this problem
- Students never need to write down anything they don't understand now
Main points of TSRJ

• *Multi-language approach* to CS1/CS2
  – Start in Scheme, develop concepts & habits
  – Switch to Java after 2-4 months

• Step-by-step *design recipe* in both languages
  – concrete questions & products at each step
  – test-first methodology
  – strong emphasis on data types
  – *shape of data determines shape of code & test cases*

• DrScheme *development environment*
  – beginner-friendly
  – interactive testing and experimentation
  – enforces *language subsets* w/appropriate messages
Let’s try DrScheme

- DrScheme
- Interactions window (bottom half)
- Literal expressions
  - 7
  - "hello world"
  - *copy and paste picture from Web browser*
  - true
Scheme syntax

- All expressions fully parenthesized; no order of operations
- All functions (built-in and user-defined) are prefix, inside parentheses
- Arity 1, 2, 3, 4, … as appropriate

Example: 3 + 4 * 5 + 6 becomes
(+ 3 (* 4 5) 6)

Try this in Interactions window
Definitions window

- Editable and savable
- "Run" button evaluates all expressions in order, replacing old Interactions
- "Step" button does same in single steps
- Try
  7
  "hello world"
  copied image from Web
  (+ 3 (* 4 5) 6)
Variable definitions

• (define bignum 1234567890)
• Try (* bignum bignum bignum)
• Note no declared data type; *data* have types, *variables* don't
Function definitions

• (define (cube num)
  (* num num num))

• Try (cube 4)
  (cube bignum)
  (cube (cube bignum)), etc.

• Note no declared parameter or return types; again, data have types

• Try (cube "hello world")
In a real class I would cover…

• function composition and re-use
• functions on images (using `image-beside`, `image-above`, `reflect-vert`, `reflect-horiz`, `rotate-cw`, `overlay`, `ellipse`, `rectangle`, etc.)
• functions on strings (using `string-length`, `string=?`, etc.)
• auxiliary/"helper" functions
• writing graphic animations (using provided package; students write event handlers as model->view and model->model functions)
Design recipes

We could write `cube` in an ad-hoc way, but in general we use…

**Step-by-step recipes** to get from English-language description to working, tested code

One recipe for data types, one for functions, one for abstractions, …
How to accomplish anything

1) Figure out what you want to do
2) Do it
3) Check that you did it right
How to write a function/method

- Figure out what you want to do
  - **Contract**: Specify name, inputs, outputs
  - **Data definition**: Identify any new data types in the problem
  - **Examples**: Write test cases with expected answers
- Do it
  - **Template/inventory**: Write boilerplate code based on data types
  - **Body**: Fill in details based on problem to solve
- Check that you did it right
  - **Testing**: Run test cases
The design recipe in practice

contract:
  ; cube : number(x) -> number

data definition: problem only involves numbers, which are predefined

examples:
  (cube 0) "should be 0"
  (cube 2) "should be 8"
  (cube -3) "should be -27"
  (cube 1234567890) "should be about 30 digits"
The design recipe in practice

Aside: There are spiffier ways to do the testing, e.g.
(check-expect (cube 0) 0)
(check-expect (cube 2) 8)
(check-expect (cube -3) -27)
(generate-report)

produces a window labeled “Test Results”:

Recorded 3 checks. 2 checks failed.
check failed at line 6 column 0
  Actual value 4 differs from 8, the expected value.
check failed at line 7 column 0
  Actual value 9 differs from -27, the expected value.
The design recipe in practice

template/inventory: (inserted before examples)
  (define (cube x)
    ; x a number
  )

body: (fill in something real, using expressions from inventory)
  (define (cube x)
    ; x a number
    (* x x x)
  )

testing: hit "Run" and see whether answers match
The result

; cube : number(x) -> number

(define (cube x)
    ; x a number
    (* x x x))

"Examples of cube:

(cube 0) "should be 0"
(cube 2) "should be 8"
(cube -3) "should be -27"
(cube 1234567890) "should be about 30 digits"
Exercise

• Write a function named `two-roots?` which takes in 3 numbers `a`, `b`, and `c` and tells whether `(b^2 - 4ac) > 0`

• Some built-in functions you’ll need:
  * : number number … -> number
  - : number number number … -> number
  > : number number number … -> boolean
Design recipes as pedagogy

• Note "test-first" methodology (a la XP); identify special cases before writing code
• Use as grading rubric: partial credit for each step
• First steps are in comments; non-threatening, avoids "blank page syndrome"
• Each step has concrete questions and concrete products
• I don't help with step N until I see step N-1
• Same steps apply in Java, C++, etc. (but more complicated)
• Know-it-all students usually try to skip the recipe… until Chapter 12 …
The Shapes problem, our way

• Function contract: 
  contains? takes in a shape and posn and returns a boolean

Product:

; contains? : shape posn -> boolean
The Shapes problem, our way

• Data definition:
  • A shape is either a circle or a rectangle
  • A circle has a posn (center) and a number (radius)
  • A rectangle has a posn (top-left) and two numbers (width and height)
• Product:
  ; A shape is either a circle or a rectangle.
  (define-struct circle [center radius])
  (define-struct rectangle [top-left width height])
The Shapes problem, our way

• Examples:
  • (define c (make-circle (make-posn 10 5) 5))
  • (contains? c (make-posn 10 5)) “should be true”
  • (contains? c (make-posn 11 3)) “should be true”
  • (contains? c (make-posn 6 9)) “should be false”
  • (contains? c (make-posn 6 8)) “should be false”

  • (define r (make-rectangle (make-posn 3 5) 7 3))
  • ; 17 test cases on r omitted
The Shapes problem, our way

• Function template/inventory:
  (define (contains? s p)
    ; s    a shape, i.e. either circle or square
    ; p    a posn, which has two numbers x & y
  )
The Shapes problem, our way

- Function template:
  (define (contains? s p)
    ; s a shape, i.e. either circle or square
    ; p a posn, which has two numbers x & y
    ; Since s is one of two possibilities, 2-way conditional:
    (cond [(circle? s) …]
             [(rectangle? s) …]
    )
  )
The Shapes problem, our way

• Function body: since each case is fairly complex, we’ll write two helper functions

(define (contains? s p)
  ; s          a shape, i.e. either circle or square
  ; p          a posn, which has two numbers x & y
  (cond [(circle? s) (circle-contains? s p)]
        [(rectangle? s) (rect-contains? s p)]
    )
)

; circle-contains? : circle posn -> boolean
; examples copied from contains?
; rect-contains? : rectangle posn -> boolean
; examples copied from contains?
The Shapes problem, our way

- Function template for circle-contains?:
  (define (circle-contains? c p)
    ;; c                a circle, which has a posn(center) and a number(radius)
    ;; p                a posn, which has two numbers x & y
    ;; (circle-center c) a posn
    ;; (circle-radius c) a number
    ;; (posn-x p)       a number
    ;; (posn-y p)       a number
  )
The Shapes problem, our way

- Function body for circle-contains?:

```
(define (circle-contains? c p)
  ; c                a circle, which has a posn(center) and a number(radius)
  ; p                a posn, which has two numbers x & y
  ; (circle-center c) a posn
  ; (circle-radius c) a number
  ; (posn-x p)       a number
  ; (posn-y p)       a number
  (< (dist (circle-center c) p) (circle-radius c))
)
```

; Another helper function:
; dist : posn posn -> number
Exercise

; dist : posn posn -> number
(define (dist here there)
  ; here                   a posn, which has 2 numbers x & y
  ; there                   another posn
  ; (posn-x here)          a number
  ; (posn-y here)          a number
  ; (posn-x there)         a number
  ; (posn-y there)         a number
)

You’ll need
  + : number number … -> number
  - : number number … -> number
  sqr : number -> number (or can do with just * )
  sqrt : number -> number
My answer

- Function body for dist:

(define (dist here there)
  ; here a posn, which has 2 numbers x & y
  ; there another posn
  ; (posn-x here) a number
  ; (posn-y here) a number
  ; (posn-x there) a number
  ; (posn-y there) a number
  ; (- (posn-x here) (posn-x there)) a number
  ; (- (posn-y here) (posn-y there)) a number
  (sqrt (+ (sqr (- (posn-x here) (posn-x there)))
         (sqr (- (posn-y here) (posn-y there)))))
)

Exercise

; rect-contains? : rectangle posn -> boolean

Some functions you’ll need:
  + : number number … -> number
  < : number number … -> boolean
  posn-x : posn -> number (extracts the x coordinate)
  posn-y : posn -> number (extracts the y coordinate)
  rectangle-top-left : rectangle -> posn
  rectangle-width : rectangle -> number
  rectangle-height : rectangle -> height
  and : boolean boolean … -> boolean

Remember, the syntax rule for calling ANY Scheme function
(built-in or user-defined) is

( function-name arg1 arg2 arg3 … )
(define-struct circle [center radius])
(define-struct rectangle [top-left width height])

(define (contains? s p)
  (cond [(circle? s) (circle-contains? s p)]
   ((rectangle? s) (rect-contains? s p)))))

(define (circle-contains? c p)
  (< (dist (circle-center c) p) (circle-radius c)))

(define (dist here there)
  (sqrt (+ (sqr (- (posn-x here) (posn-x there)))
           (sqr (- (posn-y here) (posn-y there)))))

(define (rect-contains? r p)
  (and (> (posn-x p)
          (posn-x (rectangle-top-left r)))
       (> (posn-y p)
          (posn-y (rectangle-top-left r)))
       (< (posn-x p)
          (+ (rectangle-width r)
             (posn-x (rectangle-top-left r))))
       (< (posn-y p)
          (+ (rectangle-height r)
             (posn-y (rectangle-top-left r)))))))

(define c (make-circle (make-posn 10 5) 5))
(contains? c (make-posn 10 5)) “should be true”
(contains? c (make-posn 11 3)) “should be true”
(contains? c (make-posn 6 9)) “should be false”
(contains? c (make-posn 6 8)) “should be false”

(define r (make-rectangle (make-posn 3 5) 7 3))
; 17 test cases on r omitted
Translating Scheme to Java

; A shape is either a circle or a rectangle.

; A circle has a posn(center) and a number(radius)

(define-struct circle [center radius])

; A rectangle has a posn(top-left) and two numbers (width and height)

(define-struct rectangle [top-left width height])

interface Shape {
}

class Circle implements Shape {
  private Posn center;
  private double radius;
  public Circle (Posn c, double r) {
    this.center = c;
    this.radius = r;
  }
}

class Rectangle implements Shape {
  private Posn topLeft;
  private double width, height;
  public Rectangle (Posn tl, double w, double h) {
    this.topLeft = tl;
    this.width = w;
    this.height = h;
  }
}
Translating Scheme to Java

(define (contains? s p)
  (cond [(circle? s) (circle-contains? s p)]
        [(rectangle? s) (rect-contains? s p)]))

(define (circle-contains? c p)
  (< (dist (circle-center c) p) (circle-radius c)))

(define (dist here there)
  (sqrt (+ (sqr (- (posn-x here) (posn-x there)))
           (sqr (- (posn-y here) (posn-y there))))))

interface Shape {
  public boolean contains (Posn p);
}

class Circle implements Shape {
  ...
  public boolean contains (Posn p) {
    return dist(this.center,p) < this.radius;
  }
  
  private static double dist(Posn here, Posn there) {
    double xdiff = here.getX() - there.getX();
    double ydiff = here.getY() - there.getY();
    return Math.sqrt(xdiff * xdiff + ydiff * ydiff);
  }
}
Translating Scheme to Java

(define (rect-contains? r p)
  (and (> (posn-x p)
           (posn-x (rectangle-top-left r)))
       (> (posn-y p)
           (posn-y (rectangle-top-left r)))
       (< (posn-x p)
           (+ (rectangle-width r)
               (posn-x (rectangle-top-left r))))
       (< (posn-y p)
           (+ (rectangle-height r)
               (posn-y (rectangle-top-left r)))))

class Rectangle implements Shape {
... 
  public boolean contains (Posn p) {
    return
      (p.getX() > this.topLeft.getX()) &&
      (p.getY() > this.topLeft.getY()) &&
      (p.getX() < this.width + this.topLeft.getX()) &&
      (p.getY() < this.height + this.topLeft.getY());
  }
}
Design recipe for data types

• Are there only numbers, words, pictures, true/false,…?
  – Use primitive datatypes

• Is the object made up of several parts?
  – Define a class of data with one field for each part

• Does one of the fields represent an object in another class?
  – Use composition

• Is the data type one of several choices?
  – Define a union with a common interface
Example: Shapes

- Are there only numbers, words, pictures, true/false,…?
  - radius of Circle, width & height of Rectangle are numbers
- Is the object made up of several *parts*?
  - Circle has radius & center, Rectangle has top-left, width, & height
- Does one of the fields represent an object in another class?
  - center of Circle is a Posn, top-left of Rectangle is a Posn
- Is the data type one of several *choices*?
  - a Shape is either a Circle or a Rectangle
Extending the example

- A Shape is either a Circle, a Rectangle, or a Union.
- A Circle has a center (of type Posn) and a radius (of type number).
- A Rectangle has a top-left (of type Posn), a width and a height (both of type number).
- A Union has two Shapes (named shape-a and shape-b).
- Define a function which tells whether a given Posn is inside a given Shape.
What changes?

- Data definition:
  - A shape is either a circle, a rectangle, or a union.
  - A circle has a posn (center) and a number (radius)
  - A rectangle has a posn (top-left) and two numbers (width and height)
  - A union has two shapes (shape-a and shape-b)
- Product:
  ; A shape is either a circle, a rectangle, or a union.
  (define-struct circle [center radius])
  (define-struct rectangle [top-left width height])
  (define-struct union [shape-a shape-b])
The Shapes problem, our way

Additional examples:

(define c (make-circle (make-posn 10 5) 5))
(define r (make-rectangle (make-posn 3 5) 7 3))
(define both (make-union c r))
(contains? both (make-posn 11 5)) “should be true”
   ; in circle, not rectangle
(contains? both (make-posn 7 7)) “should be true”
   ; in both circle and rectangle
(contains? both (make-posn 4 7)) “should be true”
   ; in rectangle but not circle
(contains? both (make-posn 2 7)) “should be false”
The Shapes problem, our way

• Function template:
  (define (contains? s p)
   ; s a shape, i.e. either circle, square, or union
   ; p a posn, which has two numbers x & y
  )
The Shapes problem, our way

- Function template:
  (define (contains? s p)
    ; s    a shape, i.e. either circle or square
    ; p    a posn, which has two numbers x & y
    ; Since s is one of 3 possibilities, 3-way conditional:
    (cond [(circle? s) …]
          [(rectangle? s) …]
          [(union? s) …]
          )
  )
  )
The Shapes problem, our way

(define (contains? s p)
  ; s          a shape, i.e. either circle, square, or union
  ; p          a posn, which has two numbers x & y
  (cond [(circle? s)         (circle-contains? s p)]
        [(rectangle? s)   (rect-contains? s p)]
        [(union? s)         ; (union-shape-a s)         a shape
            ; (union-shape-b s)         a shape
            [(contains? (union-shape-a s) p)   a boolean
                ; (contains? (union-shape-b s) p)   a boolean
        )]
  )
  )
)
The Shapes problem, our way

(define (contains? s p)
  ; s a shape, i.e. either circle, square, or union
  ; p a posn, which has two numbers x & y
  (cond [(circle? s) (circle-contains? s p)]
        [(rectangle? s) (rect-contains? s p)]
        [(union? s)
         ; (union-shape-a s) a shape
         ; (union-shape-b s) a shape
         ; (contains? (union-shape-a s) p) a boolean
         ; (contains? (union-shape-b s) p) a boolean
         (or (contains? (union-shape-a s) p)
             (contains? (union-shape-b s) p))
        ])
  )
)
Translating Scheme to Java

; A union has two shapes: shape-a and shape-b
(define-struct union [shape-a shape-b])

(define (union-contains? r p)
  (or (contains? (union-shape-a r) p)
      (contains? (union-shape-b r) p)))

class Union implements Shape {
    private Shape shapeA, shapeB;

    public Union (Shape a, Shape b) {
        this.shapeA = a;
        this.shapeB = b;
    }

    public boolean contains (Posn p) {
        return
            this.shapeA.contains(p) ||
            this.shapeB.contains(p);
    }
}
Another extension

• Define a function which operates on a list of arbitrarily many **Shapes** and returns how many of them are **Circles**.

• Requires defining "list of shapes" data type.
In Java, using built-in ArrayLists...

```java
import java.util.ArrayList;

public class ShapeArrayList
{
    private ArrayList<Shape> data = new ArrayList<Shape>();

    public void add (Shape newShape)
    {
        this.data.add (newShape);
    }

    public int countCircles ()
    {
        int howMany = 0;
        for (Shape currentShape : this.data)
        {
            if (currentShape instanceof Circle)
            {
                ++howMany;
            }
        }
        return howMany;
    }
}
```

```java
public class ShapeArrayListTest extends junit.framework.TestCase
{
    public void testCountCircles ()
    {
        ShapeArrayList list0 = new ShapeArrayList();
        ShapeArrayList listD = new ShapeArrayList();
            listD.add (new Rectangle(new Posn(1,1),1,1));
        ShapeArrayList listC = new ShapeArrayList();
            listC.add (new Circle(new Posn(1,1),1));
        ShapeArrayList listDC = new ShapeArrayList();
            listDC.add(new Rectangle(new Posn(1,1),1,1));
        ShapeArrayList listCD = new ShapeArrayList();
            listCD.add(new Circle(new Posn(1,1),1,1));
        ShapeArrayList listDC = new ShapeArrayList();
            listDC.add(new Rectangle(new Posn(1,1),1,1));
        ShapeArrayList listCD = new ShapeArrayList();
            listCD.add(new Circle(new Posn(1,1),1,1));
        ShapeArrayList bigList = new ShapeArrayList();
            bigList.add(new Circle(new Posn(1,1),1,1));
        bigList.add(new Rectangle(new Posn(1,1),1,1,1));
        bigList.add(new Circle(new Posn(1,1),1,1,1));
        bigList.add(new Rectangle(new Posn(1,1),1,1,1));
        bigList.add(new Circle(new Posn(1,1),1,1,1));
        bigList.add(new Rectangle(new Posn(1,1),1,1,1));

        this.assertEquals("list 0", 0, list0.countCircles());
        this.assertEquals("list D", 0, listD.countCircles());
        this.assertEquals("list C", 1, listC.countCircles());
        this.assertEquals("list DC", 1, listDC.countCircles());
        this.assertEquals("list CD", 1, listCD.countCircles());
        this.assertEquals("big list", 4, bigList.countCircles());
    }
}
```
Or if we wanted to implement linked lists ourselves...

```java
public class ShapeNullList {
    private class Node {
        Shape data;
        Node next;
        Node(Shape data, Node next) {
            this.data = data;
            this.next = next;
        }
    }
    private Node head = null;
    public void add (Shape newShape) {
        this.head = new Node(newShape, this.head);
    }
    public int countCircles () {
        Node current = head;
        int howMany = 0;
        while (current != null) {
            if (current.data instanceof Circle) {
                ++howMany;
            }
            current = current.next;
        }
        return howMany;
    }
}
```

basically the same testing code as before
What's wrong with these?

- **ArrayList version:**
  - "for" or "for each" loop (new syntax rule, complex semantics)
  - mutation/assignment (new syntax rule, complex semantics)

- **Null-terminated-linked-list version:**
  - inner class (new syntax rule, complex semantics)
  - “while” loop (new syntax rule, complex semantics)
  - mutation/assignment (new syntax rule, complex semantics)

- They're *ad-hoc*, not “obvious”; need either
  - stroke of brilliance, or
  - pre-written utility class
Let's develop it our way

• **Function contract:**
  – ; **count-circles** takes a shape-list and returns a number.

• **Data definition:**
  – A shape-list is either empty or not.
  – An empty shape-list has no parts.
  – A non-empty shape-list has a "first" shape and "the rest", which is a shape-list.
  – Product: none if we use built-in “list” data type
  – If we want to define lists ourselves,
    ; A shape-list is either an esl or a nesl
    (define-struct esl ())
    (define-struct nesl (first rest))
Our way: examples

(define c (make-circle (make-posn 1 1) 1))
(define r (make-rectangle (make-posn 1 1) 1 1))

(count-circles empty) “should be 0”
(count-circles (list r) “should be 0”
(count-circles (list c) “should be 1”
(count-circles (list r c) “should be 1”
(count-circles (list c r) “should be 1”
(count-circles (list c r c c r c)) “should be 4”
Our way: function template

(define (count-circles shapes)
  ; shapes         a list of shapes, i.e. either
  ; empty          or  (cons shape list-of-shapes)
Our way: function template

(define (count-circles shapes)
  ;; shapes         a list of shapes, i.e. either
  ;; empty or (cons shape list-of-shapes)
  (cond [(empty? shapes) …]
        [(cons? shapes) …])
))
Our way: function template

(define (count-circles shapes)
  ; shapes         a list of shapes, i.e. either
  ; empty   or    (cons shape list-of-shapes)
  ; Since there are two cases, 2-way conditional:
  (cond [(empty? shapes) …]
    [(cons? shapes)
      [; (first shapes) a shape, i.e. circle or rect
       ; (rest shapes) a list of shapes
       ]
     ]
  ))
Our way: function template

(define (count-circles shapes)
  ; shapes a list of shapes, i.e. either
  ; empty or (cons shape list-of-shapes)
  ; Since there are two cases, 2-way conditional:
  (cond [(empty? shapes) …]
        [(cons? shapes)
          ; (first shapes) a shape, i.e. circle or rect
          ; (rest shapes) a list of shapes
          ; (count-circles (rest shapes)) a number
        ]
  ))
Our way: function template

(define (count-circles shapes)
  ; shapes        a list of shapes, i.e. either
  ; empty  or    (cons shape list-of-shapes)
  ; Since there are two cases, 2-way conditional:
  (cond [(empty? shapes) …]
        [(cons? shapes)
          [(cons? shapes)
            ; (first shapes)    a shape
            ; (rest shapes)    a list of shapes
            ; (count-circles (rest shapes))      a number
            (cond [(circle? (first shapes)) …]
                  [(rectangle? (first shapes)) …]])
          ]))
Our way: function body

(define (count-circles shapes)
  ; shapes         a list of shapes, i.e. either
  ; empty   or   (cons shape list-of-shapes)
  ; Since there are two cases, 2-way conditional:
  (cond [(empty? shapes) 0]
    [(cons? shapes)
      [(first shapes) a shape
       (rest shapes) a list of shapes
       (count-circles (rest shapes)) a number
       (cond [(circle? (first shapes))
                (+ 1 (count-circles (rest shapes)))]
             [(rectangle? (first shapes))
                (count-circles (rest shapes))])
      ]
    ]))
)
The complete solution in Scheme

; count-circles : shape-list -> number
(define (count-circles shapes)
  (cond [(empty? shapes) 0]
        [(cons? shapes)
         (cond [(circle? (first shapes))
                  (+ 1 (count-circles (rest shapes)))]
               [(rectangle? (first shapes))
                (count-circles (rest shapes))])]
      )
  )
  
(count-circles empty) “should be 0”
(define c (make-circle (make-posn 1 1)
                      1))
(define r (make-rectangle (make-posn 1
                        1) 1 1))
(count-circles (list r) “should be 0”
(count-circles (list c) “should be 1”
(count-circles (list r c) “should be 1”
(count-circles (list c r) “should be 1”
(count-circles (list c r c r c)) “should be 4”
Or, defining lists ourselves

(define-struct esl [])
(define-struct nesl [first rest])

; count-circles : shape-list -> number
(define (count-circles shapes)
  (cond [(esl? shapes) 0]
        [(nesl? shapes)
         (cond [(circle? (nesl-first shapes))
                (+ 1 (count-circles (nesl-rest shapes)))]
               [(rectangle? (nesl-first shapes))
                (count-circles (nesl-rest shapes))])]
      ))

(count-circles (make-nesl r (make-esl))) “should be 0”
(define r (make-rectangle (make-posn 1 1) 1 1))
(count-circles (make-nesl c (make-esl))) “should be 1”
(count-circles (make-nesl c (make-nesl r (make-esl))) “should be 1”
(count-circles (make-nesl c (make-nesl r (make-nesl c (make-nesl r (make-nesl c (make-esl))))))) “should be 4”
Comments

• No new syntax rules (assuming conditionals and structs already taught) or mutation

• Functional program is easier to test than a stateful program

• Recursion has "sneaked in" as the obvious application of already-learned coding patterns

• The recursion is safe: if you follow the already-learned coding patterns, it can't go infinite
Exercise

Write a function named **add-up** that takes in a list of numbers and returns their sum.

• Built-ins you’ll need:
  – empty       a constant for the empty list
  – empty? : anything -> boolean   Is it the empty list?
  – cons? : anything -> boolean   Is it a non-empty list?
  – cons : object list -> non-empty list
  – list : object … -> list
  – first : non-empty-list -> object
  – rest : non-empty-list -> list
Exercise

Write a function named `sort` that takes in a list of numbers and returns a list of the same numbers in increasing order.

Hint: you’ll need an auxiliary function, but you don’t need to know any sorting algorithms; just follow the recipe!

Hint: if you’re stuck for the body, pick a non-trivial test case & annotate inventory with values for that example. Also “right answer”.
Translating Scheme to Java

; a shape-list is either an esl or an nesl
; An ESL has no parts.
(define-struct esl [])

; An NESL has a shape (first)
; and a shape-list (rest).
(define-struct nesl [first rest])

interface ShapeList {}

class ESL implements ShapeList {}

class NESL implements ShapeList {
    private Shape first;
    private ShapeList rest;
    public NESL(Shape first, ShapeList rest) {
        this.first = first;
        this.rest = rest;
    }
}
Translating Scheme to Java

; count-circles : shape-list -> number

(define (count-circles shapes)
  (cond
   [(esl? shapes)
    ; ... ]
   [(nesl? shapes)
    ; (nesl-first shapes) a shape
    ; (nesl-rest shapes) a shape-list ]))

interface ShapeList {
  int countCircles ();
}

class ESL implements ShapeList {
  public int countCircles () {
    // ...
  }
}
class NESL implements ShapeList {
  // instance variables & constructor here
  public int countCircles () {
    // this.first a Shape
    // this.rest a ShapeList
  }
}
Translating Scheme to Java

(define (count-circles shapes)
  (cond
    [(esl? shapes) ; ...
     ]
    [(nesl? shapes]
     ; (count-circles (nesl-rest shapes))
     ; a number
     (cond
      [(circle? (nesl-first shapes)) ; ...
       ]
      [(rectangle? (nesl-first shapes)) ; ...
       ]
    ]))

class ESL implements ShapeList {
  public int countCircles () {
      // ...
  }
}

class NESL implements ShapeList {
  // instance variables & constructor here
  public int countCircles () {
      // this.rest.countCircles() int
      if (this.first instanceof Circle) ...
      else ...
  }
}

// Could have used an isCircle() rather than
// instanceof, or could have added
// another polymorphic method to
Shape/Circle/Rectangle
Translating Scheme to Java

(define (count-circles shapes)
  (cond
    [(esl? shapes) 0]

    [(nesl? shapes)
      (cond
        [(circle? (nesl-first shapes))
          (+ 1 (count-circles (nesl-rest shapes)))]
        [(rectangle? (nesl-first shapes))
          (count-circles (nesl-rest shapes))]
      )]
    )])

class ESL implements ShapeList {
  public int countCircles () {
    return 0;
  }
}

class NESL implements ShapeList {
  // instance variables & constructor here
  public int countCircles () {
    if (this.first instanceof Circle)
      return 1 + this.rest.countCircles();
    else
      return this.rest.countCircles();
  }
}
The full Java program

interface ShapeList
{
    int countCircles ();
}
class ESL implements ShapeList
{
    public int countCircles () {
        return 0;
    }
}
class NESL implements ShapeList
{
    private Shape first;
    private ShapeList rest;
    public NESL (Shape first, ShapeList rest) {
        this.first = first;
        this.rest = rest;
    }

    public int countCircles () {
        if (this.first instanceof Circle) {
            return 1 + this.rest.countCircles();
        }
        else {
            return this.rest.countCircles();
        }
    }
}

public class TestShapeList extends
junit.framework.TestCase
{
    public void testCountCircles () {
        Shape r = new Rectangle (new Posn(1,1),1,1);
        Shape c = new Circle (new Posn(1,1),1);
        ShapeList list0 = new ESL();
        ShapeList listR = new NESL(r, list0);
        ShapeList listC = new NESL(c, list0);
        ShapeList listRC = new NESL(r, listC);
        ShapeList listCR = new NESL(c, listR);
        ShapeList bigList = new NESL(c,
            new NESL(r,
                new NESL(c,
                    new NESL(r,
                        new NESL(c,
                            new NESL(r,
                                new ESL())))))));
        this.assertEquals("list 0", 0, list0.countCircles());
        this.assertEquals("list D", 0, listR.countCircles());
        this.assertEquals("list C", 1, listC.countCircles());
        this.assertEquals("list DC", 1, listRC.countCircles());
        this.assertEquals("list CD", 1, listCR.countCircles());
        this.assertEquals("big list", 4, bigList.countCircles());
    }
}
Comments

• More syntax than Scheme, but again based on previously-learned coding patterns

• Where's the null? Replaced by polymorphism
  – Common beginner mistakes caught at compile-time, not run-time
  – No need to overload null to mean both an uninitialized variable and the end of a list

• Where’s the while-loop? Replaced by recursion
  – No new syntax rules, no need for mutation, easier to state invariants
Abstraction

We could write `countRectangles` in the same way, but why repeat the code?

Instead, generalize the problem to “count elements of a list that meet a certain criterion”
Design Recipe for Abstractions

1. Compare two similar pieces of code
2. Highlight the differences
3. Replace the differences by parameters
4. Design the abstraction using the parameters
5. Implement the original code in terms of the abstraction
6. Run the original tests on the new solution
Design Recipe for Abstractions

1. Compare two similar pieces of code
   - count-circles and count-rectangles

2. Highlight the differences
   - what test do you perform on (first shapes)?

3. Replace the differences by parameters
   - replace circle? and rectangle? with test?
   - add a parameter named test?
   - wait a moment: what type is test? ?
Higher-order functions

• "Counting list elements that match a criterion" is a common task.

• Scheme makes passing functions as parameters extremely easy, so…

• A Scheme programmer naturally abstracts the criterion into a parameter, thus producing a much more general and powerful function.
Higher-order functions

; count-if: (shape -> boolean) (list-of shape) -> number
(define (count-if test? items)
  (cond
   ((empty? items) 0)
   ((cons? items)
    (cond
     ((test? (first items))
      (+ 1 (count-if test? (rest items))))
     (else (count-if test? (rest items))))))))

; count-circles : list-of-shape -> number
(define (count-circles shapes)
  (count-if circle? shapes))
Higher-order functions

; count-if: (X -> boolean) (list-of X) -> number
(define (count-if test? items)
  (cond
   ((empty? items) 0)
   ((cons? items)
    (cond
     ((test? (first items))
      (+ 1 (count-if test? (rest items))))
     (else (count-if test? (rest items)))))))

; count-circles : list-of-shape -> number
(define (count-circles shapes)
  (count-if circle? shapes))
Doing this in Java

- Java doesn't allow passing a method as a parameter
- You have to package it up in an object whose class implements an interface
- But programming with higher-order functions is still a useful technique
- Get students hooked on it in Scheme, where it's easy
- Java 1.5 allows abstracting over types too
Translating Scheme to Java

; count-if:
    (X -> boolean) list-of-X ->
    number

interface Test<X> {
    boolean test (X p);
}

interface List<X> {
    int countIf (Test<X> t);
}
Translating Scheme to Java

(define (count-if test? items)
  (cond
   ((epl? items) 0)
   ((nepl? items)
    (cond
     ((test? (nepl-first items))
      (+ 1 (count-if test? (nepl-rest items))))
     (else
      (count-if test? (nepl-rest items)))))))

class EmptyList<X> implements List<X> {
  public int countIf (Test<X> t) {
    return 0;
  }
}

class NonEmptyList<X> implements List<X> {
  // instance vars & constructor here
  public int countIf (Test<X> t) {
    if (t.test(this.first))
      return 1 + this.rest.countIf(t);
    else
      return this.rest.countIf(t);
  }
}
The full Java program

interface Test<X>
{
    boolean test (X p);
}

interface List<X>
{
    int countIf (Test<X> t);
}

class EmptyList<X> implements List<X>
{
    public int countIf (Test<X> t) {
        return 0;
    }
}

class NonEmptyList<X> implements List<X>
{
    private X first;
    private List<X> rest;
    public NonEmptyList (X first, List<X> rest) {
        this.first = first;
        this.rest = rest;
    }

    public int countIf (Test<X> t) {
        if (t.test(this.first)) {
            return 1 + this.rest.countIf(t);
        } else {
            return this.rest.countIf(t);
        }
    }
}
What's OOP about?

Imperative/procedural:
verbs -> nouns

Identify important operations, then…
define data types that enable you to implement those operations

OOP:
nouns -> verbs

Identify important data types, then…
choose & implement important operations on them
Why nouns->verbs?

• Problem statements usually describe data types, not algorithms.

• Data types tend to exist in the problem domain; algorithms are generally only in the solution domain.

• e.g. "find the largest number in a list" mentions two data types, but no algorithms.
Approaches to CS1/CS2

CS1/CS2 sequence, developed in pre-OOP era: **verbs -> nouns**

CS1: learn how to code algorithms; learn a lot of language features
CS2: learn to define new data structures; start analyzing algorithms

TeachScheme!:

**nouns -> verbs**

CS1: For each data type, learn to design, test, and code algorithms involving that type
CS2: Repeat in Java; start analyzing algorithms
Who uses this approach?

- About 100 colleges and universities
- At least 200 high schools and middle schools
- Some use it for starting CS students; others in a non-major programming course
<table>
<thead>
<tr>
<th>Location</th>
<th>Host(s)</th>
<th>2007 Dates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Garden City, NY</td>
<td>Adelphi University</td>
<td>June 18 - 22</td>
</tr>
<tr>
<td>San Luis Obispo, CA</td>
<td>Cal Poly</td>
<td>June 25 - 29</td>
</tr>
<tr>
<td>Boston, MA</td>
<td>Northeastern University/WPI</td>
<td>July 23 - 27</td>
</tr>
<tr>
<td>Salt Lake City, UT</td>
<td>University of Utah</td>
<td>July 30 - August 3</td>
</tr>
</tbody>
</table>